# ADL Final Project - Reinforcement Learning With The Bullet Hell Game

I-Ping Chou
R08946006
r08946006@ntu.edu.tw

Wen-Hua Wang
R08921087
r08921087@ntu.edu.tw

Chia-Ho Tsou
R08921108
r08921108@ntu.edu.tw

Hsin-Chih Yang
R08946024
r08946024@ntu.edu.tw

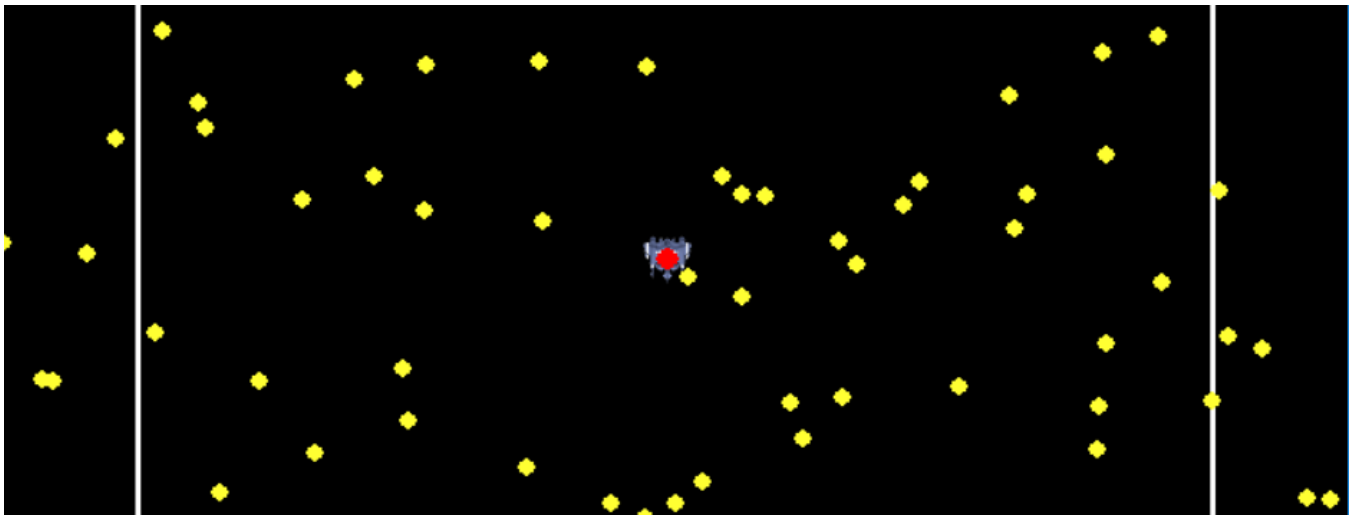

**Figure 1.** A bullet hell game made by pygame

## Abstract

Reinforcement Learning are widely used in AI game-play learning field. Model training efficiency and performance is therefore an important study. In our study, we focus on how RL models play the bull hell game. We design numerous method, such as Proximal Policy Optimization (PPO), Generalized Advantage Estimation (GAE), Soft Actor-Critic (SAC) and Rainbow DQN, and discuss each performance in order to find the best method in the bullet hell game. By observing how the model make interaction with our game environment, we modify the game design to fit the model training with well-tuned hyper-parameters on PG and DQN-based models.



*ADL 2020, June 0628, 2020,*



Source code : https://github.com/IPINGCHOU/ADL-Final-Project

## 1 Introduction

In ADL hw3, students were asked to setup a basic PG model and DQN model with several variants. By training those model on some well-known Atari games, we discovered that the model are learning policies and trying to get higher score on each game.

In our final project, we not only want to dig in more about Reinforcement Learning but try to set up our whole training process from scratch.

Instead of the game like 'Pacman' holding a constant level design, we chose bullet hell game to observe how model adapt to such kind of high randomly-generated game pattern. We implemented several PG and DQN based model variants in our work, all of them were trained with thousands of episode and the experiments results and discussion will be addressed in the following sections. In section 2, we briefly introduce our approach, mostly about the models we used. Game environment design with all function usage and game-mode selection were addressed in section 3. We separate section 4

with several subsections, in subsection 4.1 we discussed how the game design impact our model training process and the modification on it. From section 4.2 to 4.5 list out all our experiments and video demos with PG and DQN based models, includes : PPO, GAE, SAC, Dueling-DQN, Double-DQN and Rainbow. In section 5, we summarize our simulation result and lastly, the work distribution will be addressed in section 6.
Codes were pushed to our github link https://github.com/IPINGCHOU/ADL-Final-Project

## 2 Approach

### 2.1 Proximal Policy Optimization (PPO)

PPO [13] is an off-policy method to update policy $\theta$, PPO1 uses a penalty on Kullback-Leibler (KL) divergence,and $\beta$ is a hyperparameter, $\hat{A}_t$ is the estimated advantage at time t

$$L^{KLPEN}(\theta) = \tag{1}$$

$$\hat{\mathbb{E}}_t \left[ \frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \hat{A}_t - \beta \operatorname{KL}\left[\pi_{\theta_{\text{old}}}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right)\right] \right]$$

PPO2 sets hyperparameter $\varepsilon$ to clip surrogate objective.

$$L^{CLIP}(\theta) = \hat{E}_t \left[ \min\left( r_t(\theta)\hat{A}_t, \operatorname{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}_t \right) \right] \tag{2}$$

### 2.2 Generalized Advantage Estimation (GAE)

GAE [12] design a new method to estimate the advantage function that balance the variance and bias.

$$\hat{A}_t = \sum_{l=1}^{\infty} (\gamma\lambda)^l (r_t + \gamma V(s_{t+l+1}) - V(s_{t+l}))) \tag{3}$$

where $\gamma$ is discount rate, $\gamma V(s_{t+l+1}) - V(s_{t+l})$ is TD residual.

### 2.3 Soft Actor-Critic (SAC)

Other RL algorithms typically suffer from two major challenges: very high sample complexity and brittle convergence properties. Both of these challenges severely limit the applicability of such methods to complex, real-world domains. Therefore, we choose soft actor-critic (SAC)[3] [7], an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning framework.

### 2.4 Rainbow

Rainbow integrate all the following seven components into a single integrated agent, which is called Rainbow

- DQN
- Double DQN - off policy training for DQN
- Prioritized Experience Replay - Make priorities to samples
- Dueling Network - Separate output layer to advantage and value layer.
- Noisy Network - Add additional self-adaptive noise to the output layers
- Categorical DQN - Estimate the distribution of the output rather than the raw output.
- N-step Learning - Take N step for future estimation.

We implemented this method for not having a good result in both dueling-DQN and double-DQN. Rainbow combines all the benefits from each DQN variants and we hope it could bring us a better model performance.

## 3 Game Environment design

We applied the 'Pygame' package for our self-designed bullet-hell game, which allowed us to get individual states and rewards. Besides, we packed the game environment as the 'gym' which did for better function calling during model training. Our environment provided the following functions for model training. The following game design were referred to [6]

- `step`: It returns the state, reward, collision, done for each step of the game play, where collision comes to be true while the plane crashes into the bullet, and done indicates if the game is finished or not (we made a explosion sprite if the plane crashed, this can also be disable during model training).
- `reset`: It reset the whole game environment to its initial state and returns it.
- `resize_state`: We discovered that the model training speed suffers from large input image size, or causes GPU memory usage problems when training DQN with large replay buffer. Thus we designed this function for resizing the output state.
- `is_collision`: This function checks if the plane crashed, return True if any of our bullets collided with the plane.
- `is_warning_zone`:Returns the counts of the bullets which enters the warning zone.
- `ReplaySaver`: Saves every state from the model playing the game, and returns the game-play video to the desired folder as a mp4 file.

All the game parameters such as hitbox radius, bullet configs, plane configs ... etc are all set in `./game/game_config.py`, all of them are modifiable and can soon be checked by activating the game by executing the game.py under each model folder, for example : `./Rainbow_GPUbuffer/game.py`. Before the game starts, we also provide test mode, score mode, plane mode and explosion mode for no bullets, no score display, no plane sprite and no explosion sprite in game. Give our game a test by trying to dodge the bullets from 4 corners! Use arrow keys to control your plane, you will get 0.1 reward for every step you stay alive and -100 when you are killed. If the game is too hard for you, try to decrease the

FPS from `./game/game_config.py`, it can provide you with more reaction time to survive.

# 4 Experiments

## 4.1 Game Environment Modification

We discovered several game environment settings will make big difference to our model training, specially in the first wave of our game, the following subsections are the modification we made. Every model experiments were trialed under the modified game environment.

**4.1.1 Reward.** We initially set survive reward=1 and death reward=-100 for each step,but we observe DQN will overestimate the reward at the start of training.It may estimate death frame have 1 reward,and lead to all traces reward increase,so we set survive reward=0.1 and death reward=-100 to reduce the impact on overestimating.

**4.1.2 Bullet Respawn Pattern.** In our initial game settings, the bullets will aim at our plane whenever the bullets were spawned. This causes our model hard to dodge at the start of the round, which leads our model tend to make the same move in order to try its luck to escape from the first round. Thus we decided to make the bullets spawn with random angle instead of aiming at our plane. Gives our model more chance to achieve its target - dodging the bullets.
Demo: https://youtu.be/HVy39v0xVtc

**4.1.3 Bullet speed and maximum spawn counts in each steps.** Fixed bullet speed and unlimited bullet spawn in single step will also make the model hard to escape if multiple bullets just spawn at the same time with a specific angle, which somehow traps the model. We give bullets random speed to break such kinds of dilemmas. This helps the model to make priority order to incoming bullets, which increases the learning speed during training.
Before: https://youtu.be/HVy39v0xVtc
After: https://youtu.be/AycEoXTYPBk

**4.1.4 Game Border.** With thousands of our trials, the model tends to move to the border of our game surface, which suffers from the low visibility from the bullets come from the border. To solve this problem, we designed a plane-movement border to limit the plane from moving to the border of our game window. This feature makes the model able to catch every bullet insight, which actually help the model to dodge the bullets in early training stage.
Demo: https://youtu.be/jFth7WeK1H4

**4.1.5 Invincible Mode.** In our first design, we set the game to soon end if the plane crashed to a single bullet. However, we discovered that there will be only one state that model will actually get a penalty, which might slow down our learning target - 'don't touch a single bullet'. So, we decided to design an invincible mode for the model. The game will still move on til a desired game steps if the plane crashed into a bullet. This makes the model able to sample more states that received penalties, increasing the speed of learning the desired principle.
Demo: https://youtu.be/JI8DS_ocQ-Y

## 4.2 Proximal Policy Optimization (PPO)

**4.2.1 PPO1.** PPO hyper-parameters are set as table: 1
After over 6000 episode, the learning curve is shown in Figure 2: We can see this curve changes dramatically. We test this model for 100 episode,the average reward is only -84 (160 steps).

**Table 1.** PPO1 hyper-parameters settings

| Hyper-parameters | Value |
|---|---|
| PPO1 | |
| `RMSprop stepsize` | $10^{-3}$ |
| $\beta$ | 0.5 |
| $KL_{target}$ | 0.01 |
| `Discount` ($\gamma$) | 0.99 |

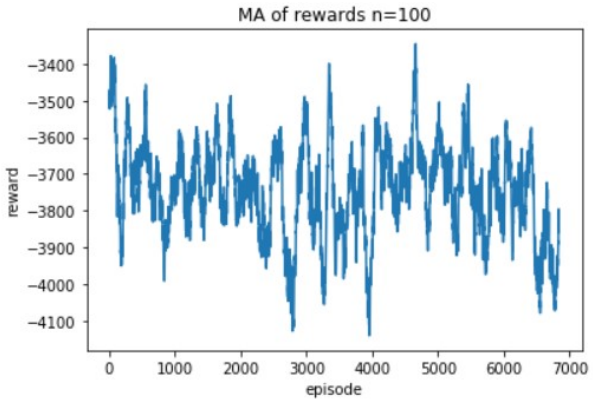


**Figure 2.** PPO1 reward plot

**4.2.2 PPO2.** We change the input state in this experiment, using the local frame which center is plane instead of full frame, and decrease the learning rate to avoid the curve dramatically changing.
After 6000 episode,the learning curve is shown in Figure 3: There is a slight growth trend in this curve,and smoother than PPO1's result.We test this model for 100 episode,the average reward is -81.8 (182 steps).
Demo: https://youtu.be/uzuPKoi3Jz4

## 4.3 Generalized Advantage Estimation (GAE)

The GAE hyper-parameters are set as table: 3
In the PG-based method, it is suffer from the reward from

**Table 2.** PPO2 hyper-parameters settings

| Hyper-parameters | Value |
|---|---|
| PPO2 | |
| RMSprop stepsize | $10^{-4}$ |
| clip ($\varepsilon$) | 0.2 |
| Discount ($\gamma$) | 0.99 |

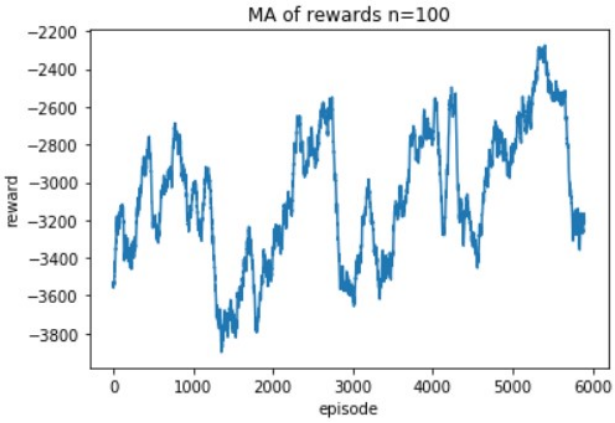


**Figure 3.** PPO2 reward plot

on-policy problem because the large variances. Therefore, GAE method offered the great estimation to fix the problem.

**Table 3.** GAE hyper-parameters settings

| Hyper-parameters | Value |
|---|---|
| GAE | |
| RMSprop stepsize | $10^{-5}$ |
| Batch size | 8 |
| $\lambda$ | 0.99 |
| Discount rate ($\gamma$) | 0.99 |

After over 40000 episode, the reward curve is shown in Figure 4:
Demo: https://youtu.be/MemrVoKSkIU

### 4.4 Dueling DQN and Soft Actor-Critic (SAC)

Due to the result that we've tested on the initial environment settings is not satisfactory, we are curious about whether the model really pay attention to the position of the plane and bullets while making decisions. Therefore, we add a saliency map[2] on every state (frame) while testing. Link below is the video showing that network cares about which part of the state when making actions. We can found that network really know where the plane and bullets are. Hence, we consider that there must be problems in game mechanism and

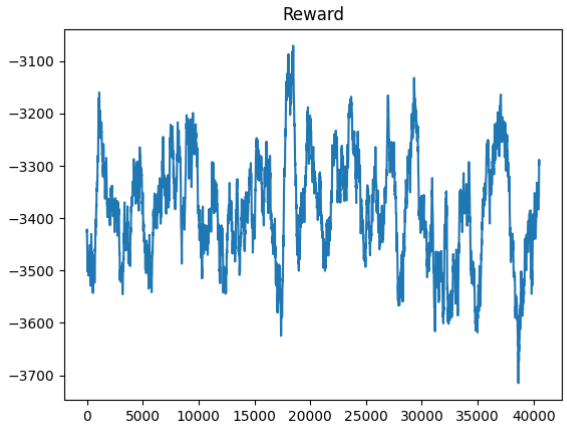


**Figure 4.** GAE reward plot

modify the environment settings as mentioned in 4.1. In this section, we compare the performances of two RL algorithms on the basis of the new environment.

Demo : https://youtu.be/IeeblNF2MbI

**4.4.1 Dueling DQN.** Table 4 shows the hyper-parameters, and the model is trained on invincible mode (i.e., 1000 steps episode) for 1000 episodes. Figure 5 shows the learning curve.

**Table 4.** Dueling DQN hyper-parameters

| Hyper-parameters | Value |
|---|---|
| Optimizer | RMSprop |
| Learning rate | $10^{-4}$ |
| Batch size | 2048 |
| Discount ($\gamma$) | 0.99 |

**4.4.2 SAC.** Table 5 shows the hyper-parameters, and the model is trained on invincible mode (i.e., 1000 steps episode) for 1000 episodes. Figure 6 shows the learning curve.

**Table 5.** SAC hyper-parameters

| Hyper-parameters | Value |
|---|---|
| Optimizer | Adam [9] |
| Learning rate | $3 * 10^{-4}$ |
| Batch size | 128 |
| Discount ($\gamma$) | 0.99 |

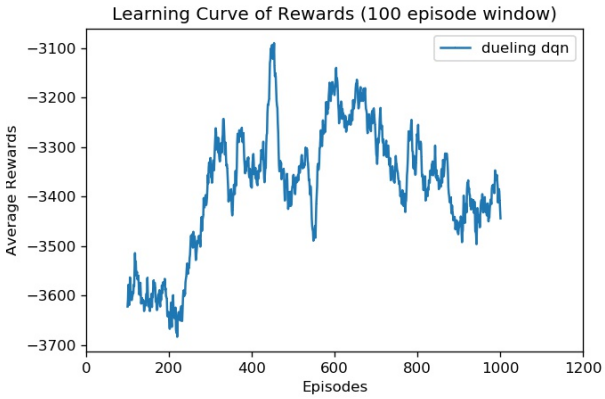


**Figure 5.** Dueling DQN reward plot

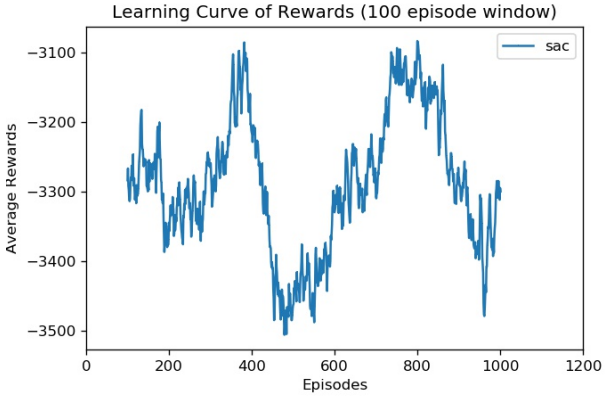


**Figure 6.** SAC reward plot

#### 4.4.3 Comparison and Testing Result.

First, we found that during training, both learning curves have ascending trends, but descend dramatically at some episodes. Second, the peak of SAC learning curve is slightly higher than Dueling DQN, so we suppose that SAC might have better performance than Dueling DQN.

Then we test both models on normal mode (i.e., collision leads to the end of an episode) for 30 episodes respectively. Result shows that the best reward with Dueling DQN is -70.6, and the average reward of 30 episodes is -86.1. On the other hand, the best reward with SAC is -33.7, and average reward of 30 episodes is -83.6. Links below are the best episode gameplay videos.

Dueling DQN : https://youtu.be/Z4-0nliK-HY
SAC : https://youtu.be/VXaHMcTBq8M

According to the testing result, we consider that SAC is more suitable for such tasks like bullet hell game than Dueling DQN.

### 4.5 Rainbow

Besides PG-based RL algorithms, we also tried Rainbow because it's the state-of-the-art of DQN-based models. The followings will mainly describe how we tuned our hyperparameters for each DQN variant in Rainbow. All the codes are refer to [5] and paper [8]

#### 4.5.1 Feature extraction - CNN with FPN.

We input a size $200 \times 200$ state image for each state, which needs several convolution layers to extract the image information. This might cause the information loss in such design, thus we applied the FPN(Feature Pyramid Networks) to prevent such loss occurs. The model architecture also show in figure 7. The code refered to [4] and paper [10]

#### 4.5.2 Noisy Net.

The noisy net provides an adoptive way to give noises to model outputs. By simply applying an additional gaussian noise on the output layer, the model will be able to ignore the noise along the training, meanwhile providing a reasonable exploration exploitation in the beginning of the training. However, we discovered that the gradient of the parameters vanished during the training which caused the noisy net to NOT update at all. Unfortunately we can't fix this problem, thus we returned to the epsilon greedy method.

#### 4.5.3 Categorical DQN.

Categorical DQN assumed every action of the model should follow its own distribution. We discovered that the approximate distribution in early training stages shrink to 0 for each quantile after the sigmoid function due to large atom size. Thus we decided to lower the atom size to 51 instead of 101.

#### 4.5.4 N step info.

This trick allowed us to sum up the past rewards with an additional decay. But we disable it because this trick will aggregate the reward for future state reward estimation. With the formula below:

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} \tag{4}$$

In bullet hell game, every alive state should give the same reward, but in this formula, along the plane is alive, step n+1 will receive a higher reward than step n. Which might lead the model to do strange action. The figure **??** shows the difference before and after disable the N step information trick. Loss curve are able to converge in early training stages, and maintain a stable learning curve.

#### 4.5.5 Prioritized Experience Replay (PER).

Using a replay memory leads to design choices at two levels: which experiences to store, and which experiences to replay (and how to do so). In our bullet hell game, every bullet is randomly shot (with random speed and angle) , thus we guess the model should take more time to learn not to touch the bullets, which means the future steps should be more important. For this purpose, we slow down the importance

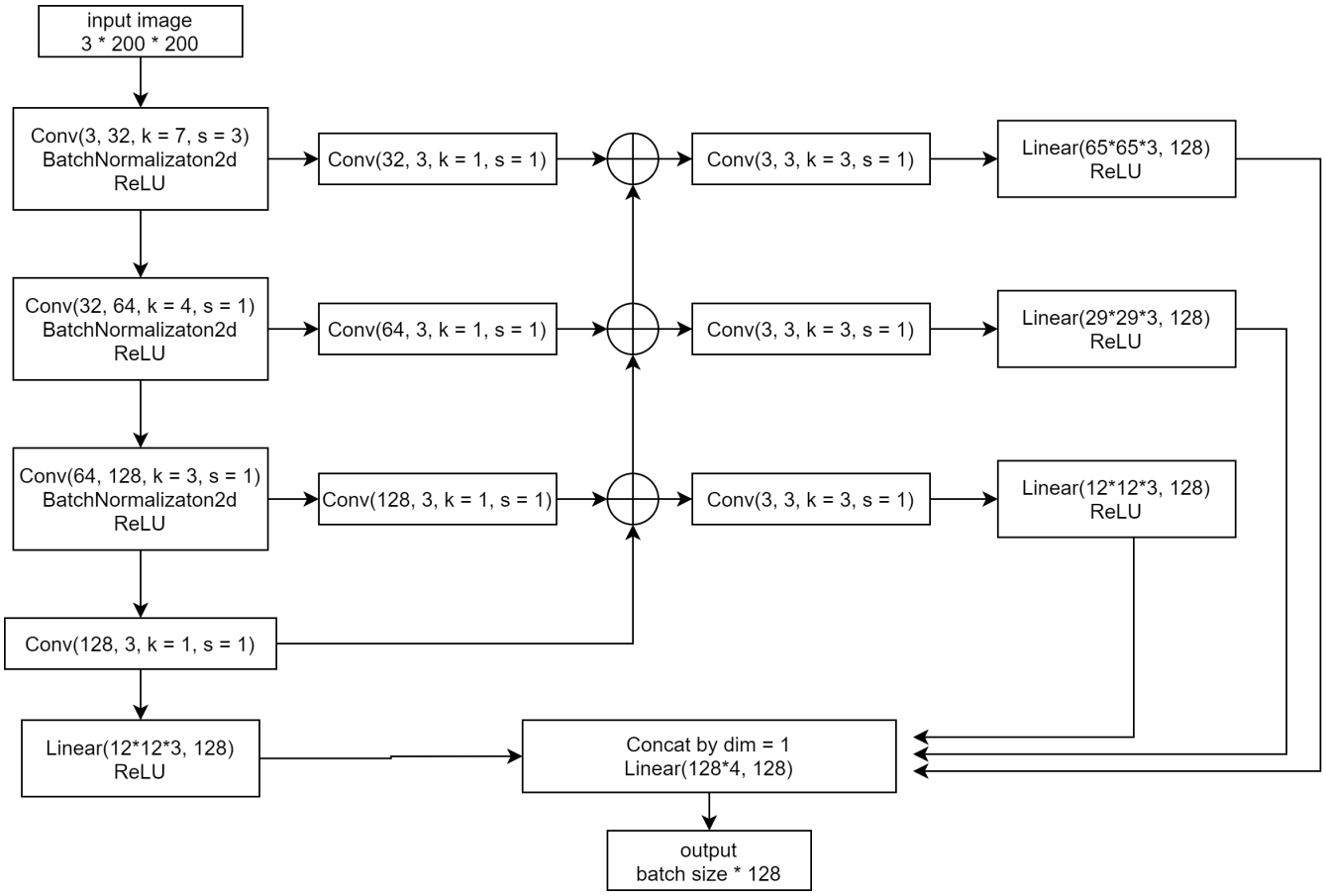


**Figure 7.** Rainbow feature extraction model

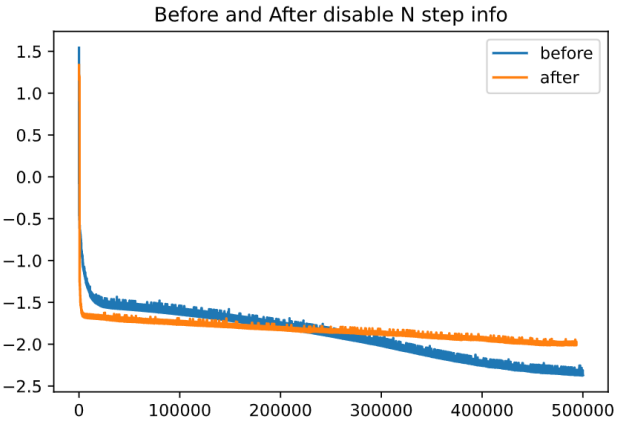


**Figure 8.** Loss curve of before and after disabled N step info plot

increasing rate to make the sample importance smoother during the training.

#### 4.5.6 Dueling and Double DQN.

DQN variants that are already mentioned in the teachers class. We didn't modify anything for this part.

#### 4.5.7 Hyper-parameters in DQN.

We found that learning rate, batch size, buffer size and target net update frequency make a big difference during training. The following tuning principle refer to [1]

- Learning Rate:
  In our case, an excessive learning rate might lead to loss explosion and conservative model learning. Conservative model learning stands for models which aren't trying to do any action but the first action it made, which causes the plane to keep bumping to the game border or stand still till the game ends.
- Batch size:
  We tried batch size from 32 to 512 and found out a small batch size will also lead to a conservative action problem, this probably due to a bigger batch size provides a smoother distribution, while small batch size gives a bad sampling result cause the model tends to do the same action in every state.

- Target update frequency:
  This might make the most impact to the model, we have tried the frequency from 10, 100, 1000 to 2000. 10 and 100 makes the training process extremely unstable, sometimes it even cause the loss to explode. While 1000 actually stabilizes the loss curve, but leads the model to repeat the same move over 1000 episodes. 2000 did the best job with a reasonable loss curve and maintain the model activity for bullet dodging.

We finally get our best experimental results with the hyper-parameter settings in Table 6.
We trained a model with over 2000 episodes that is able to get only a single hit in 500 steps, check out the demo in https://youtu.be/L5KFWL4sRiM. Or this one with 3 hits in 500 steps https://youtu.be/7Daq72GhQx8.
Also, we test out the difference between our model and random actions. In one-live mode, we plot out the distribution of rewards from both Rainbow and random in figure 9. Rainbow has a flatter distribution that random has, which means Rainbow does show better performance on dodging the bullets. Also, Rainbow reaches a longer survival time than random does.

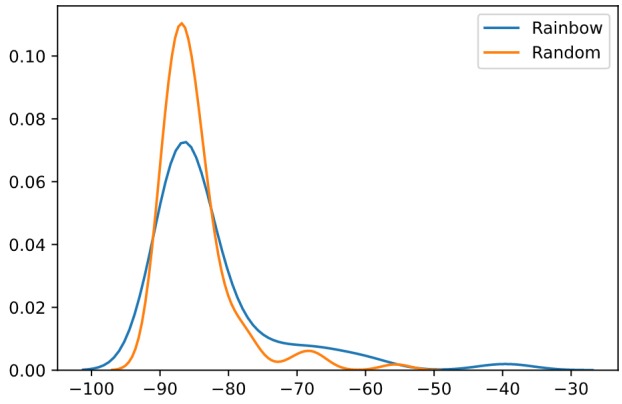


**Figure 9.** Accumulate rewards distribution with single live, Rainbow and random

## 5 Conclusion

In our result, Rainbow DQN shows a better game-play performance compare to PG based algorithms such as PPO, GAE, SAC...etc. The reason might because the high randomness of the bullet hell game, which makes PG-based algorithms hard to find a policy that best suit such kind of game environment. Meanwhile, Rainbow with numerous variants held a better adaptability on high randomness game.
Most of our PG based model turned out to repeat the same move in every state while Rainbow tries to dodge the bullets from every directions. PG might misunderstanding the game target from 'dodging the bullets' to 'find a safest position'.

**Table 6.** Rainbow best hyper-parameters settings

| Hyper-parameters | Value |
|---|---|
| DQN | |
| `Optimizer` | RMSprop [11] |
| `Memory Capacity` | 20k |
| `Batch size` | 512 |
| `Target net update frequency` | 2000 |
| `Learning rate` | $10^{-5}$ |
| Variants | |
| `Noisy Net` | disabled |
| `N step learning` | disabled |
| `PER` $\beta_{start}$ | 0.4 |
| `PER` $\beta_{steps}$ `(steps to 1, increase linearly)` | 500k |
| $\epsilon_{start}$ | 0.99 |
| $\epsilon_{end}$ | 0.05 |
| $\epsilon_{steps}$ `(steps to 1, increase linearly)` | 200k |
| `Categorical atom size` | 51 |
| `Categorical` $v_{min}$ | 0 |
| `Categorical` $v_{max}$ | 200 |

Also, the game environment design makes great impact to our model training process, even Rainbow can't get a satisfying performance in our initial game design. The bullet spawn pattern with balance wave design helps our Rainbow model eventually get a out-standing dodging performance.
To sum it up, we made a highly customized game environment by pygame package for custom model training design. By numerous trails and errors, we trained a Rainbow model that is capable of dodging bullets with only a single hit in 500 steps. Examine the difference of PG based and DQN based model differences on highly randomness game play.

## 6 Work distribution

- I-Ping Chou (R08946006):
  - Game environment design with game-play video output.
  - Game environment modifications.
  - Rainbow model design and experiments.
- Wen-Hua Wang (R08921087):
  - Design DDQN, PPO1 and fine-tune parameters.
  - Optimize reward function settings.
- Chia-Ho Tsou (R08921108):
  - Design Dueling DQN, SAC and experiment.
  - Visualization.
- Hsin-Chih Yang (R08946024):
  - Game environment modifications.
  - Design PPO2, GAE and fine-tune parameters.

## References

[1] [n.d.]. *DDQN hyperparameter tuning using Open AI gym Cartpole.* https://adgefficiency.com/dqn-tuning/

[2] [n.d.]. *Explainable DQN Pytorch.* https://github.com/vlievin/explainable-dqn-pytorch

[3] [n.d.]. *PG is all you need.* https://github.com/MrSyee/pg-is-all-you-need

[4] [n.d.]. *PyTorch-FPN.* https://github.com/bobo0810/PytorchNetHub/tree/master/FPN_pytorch

[5] [n.d.]. *Rainbow is all you need.* https://github.com/Curt-Park/rainbow-is-all-you-need

[6] [n.d.]. *Using Deep Q-Network to Learn How To Play Flappy Bird.* https://github.com/yenchenlin/DeepLearningFlappyBird

[7] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic Algorithms and Applications. arXiv:1812.05905 [cs.LG] https://arxiv.org/abs/1812.05905

[8] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. *CoRR* abs/1710.02298 (2017). arXiv:1710.02298 http://arxiv.org/abs/1710.02298

[9] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

[10] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. 2016. Feature Pyramid Networks for Object Detection. *CoRR* abs/1612.03144 (2016). arXiv:1612.03144 http://arxiv.org/abs/1612.03144

[11] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. arXiv:1609.04747 [cs.LG] https://arxiv.org/abs/1609.04747

[12] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-Dimensional Continuous Control Using Generalized Advantage Estimation. (06 2015). https://arxiv.org/abs/1506.02438

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. (07 2017). https://arxiv.org/abs/1707.06347